Augmented Binary Search Trees

Recall that binary search trees support three main dynamic-set operations: Insert, Delete, and Search. We saw in our previous lecture that these operations can run in time $O(\log n)$ for BST's that have height $O(\log n)$, such as *Red-black* trees.

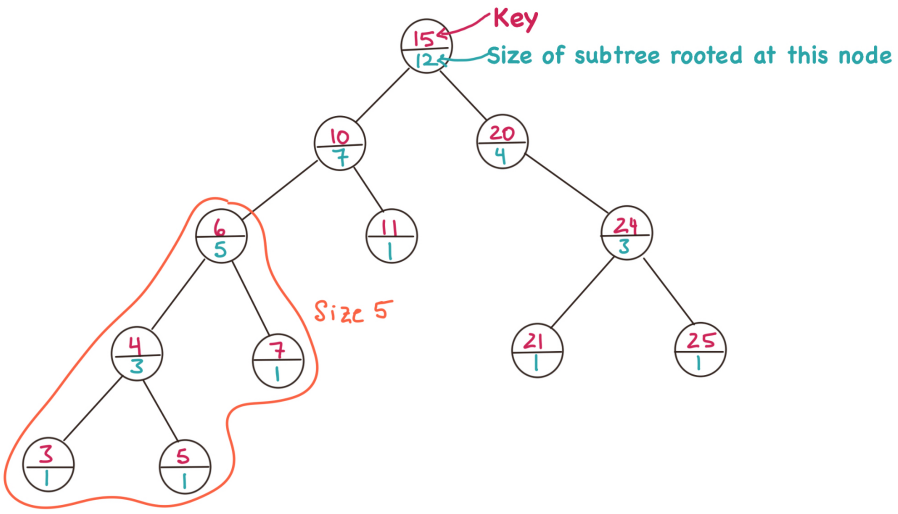
A typical BST stores a *key* in each node, which is the primary information stored by the tree. Additional information can be stored in the node object, as in *Red-black* trees, where we also stored the *color* of each node. This is usually called **augmenting** the BST with **additional information**. In this lecture, we look at further ways of augmenting a BST that then enables us to carry out the following additional operations:

- Select an element by rank : returns the item corresponding to rank k from the BST
- Returning the rank k of a given element x from the tree.
- Finding the total number of elements that lie within a specific *range*: $a \leq x \leq b$

We first look at how to augment the binary search tree by augmenting each node with its **subtree size**. This allows us to efficiently perform the above mentioned operations. Adding additional information to the binary search tree is not always easy. As soon as we add new information to the tree, then each time we do an insert/delete operation we need to *update* this additional information as well. This must be done efficiently, otherwise our augmented BST will have slower insert and delete times. The goal is to maintain the three primary operations of the balanced BST so that they continue to run in time $O(h)$.

1 Augmenting the BST with subtree sizes

In order to carry out the above three operations in a BST, we **augment** the BST by storing the **subtree** size in each node of the tree:



The implementation of the tree would now involve node objects that have an additional attribute, such as $x.size$ that refers to the subtree size of the node rooted at x. Note that nodes that are leaves have a subtree size of one, since the size of the subtree includes the node itself. The subtree size at the root is in fact the number of nodes in the entire tree.

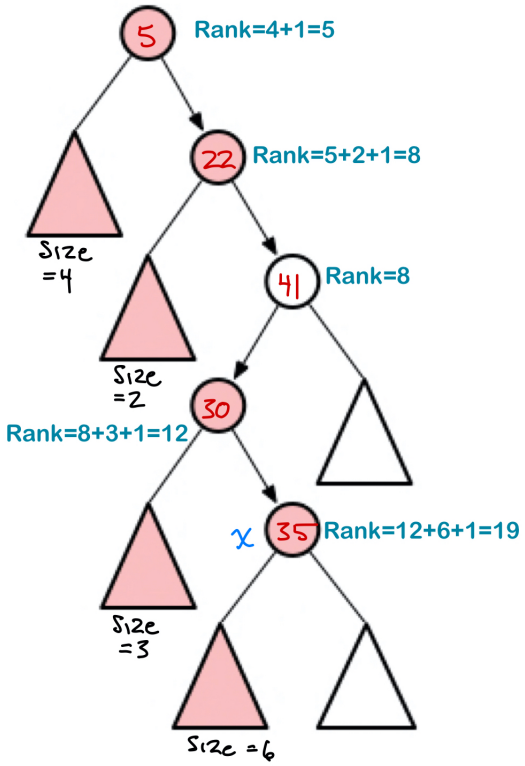
We shall now show that this *augmented* binary search tree enables us carry out our new operations.

2 Finding the Rank of an element

Suppose we would like to determine the **rank** of a particular element in the tree. In the above figure, the key 11 in the tree has rank 7 because it is the 7th smallest of all elements in the tree. Given a node x (a pointer to a node) in a BST, we would like to determine the rank of this node in the tree. The figure below is a general diagram of a BST, with the subtrees drawn as triangles. Notice that all the elements in pink are those that guaranteed to be *smaller or equal* to x. Therefore the rank of x is simply the sum of all the pink nodes. The pink nodes can be categorized as either:

- Those that are in the left subtrees on the path from the root to x
- The nodes on the path to x that preceded a right child.

Thus if we simply **count** all these nodes as we search down the tree for x, we can determine the rank of x. This is a simple update to the TREE-SEARCH algorithm from the lecture on BST. The update is to simply sum up the left-subtree sizes as we go down the tree, and also sum the nodes on the path whenever we make a turn to the right. This sum can be stored in a variable, *rank*.



In the example above, the final rank of node x is 19. The pseudocode for the Rank algorithm is shown below. The algorithm takes as input a pointer t to the root node, and x is a pointer to particular node in the tree, and assumes that x is in the tree. The result is that it returns the rank of element x.

```
RANK(t,x)
    rank = 0
    while (t ≠ x)
        if (t.key < x.key)
            if t.left ≠ NIL
                rank += t.left.size
            rank++
            t = t.right
        else t = t.left
    if t.left != NIL
        rank += t.left.size
    rank++
    Return rank
```

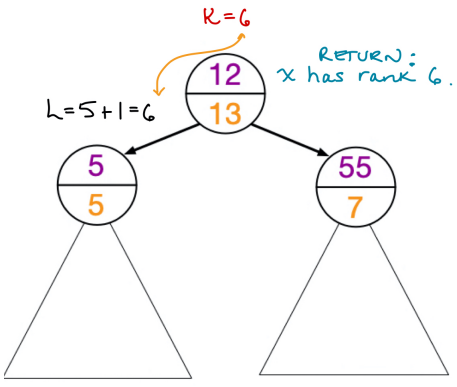
Runtime: This algorithm searches down the path from the root to x and calculates the rank as it goes. The runtime is $O(h)$. If the BST is implemented as a self-balancing tree, such as a Red-black tree or AVL tree, then the height of the tree is $O(\log n)$ and therefore $Rank(x)$ runs in time $O(\log n)$.

3 Selecting an element by rank

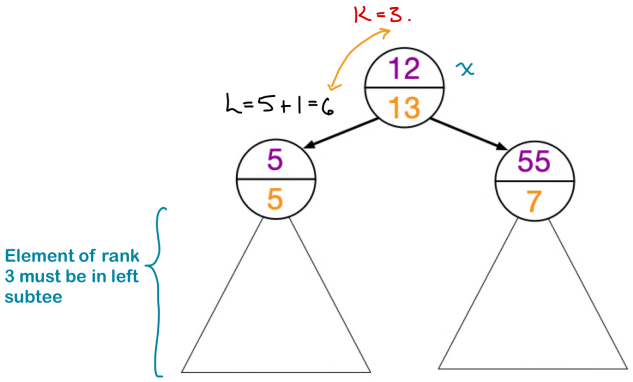
In our previous lectures, we studied the **Select** algorithm that returns the element with rank k in a set of n elements. That algorithm solved this problem for a static set of elements, in time $O(n)$. In this section, we look at how to provide a *dynamic* solution to finding the element of rank k. The word *dynamic* refers to the fact that our set S in changing every time there is an insert or delete, and the select algorithm should operate on this changing set S. We refer to this new Select algorithm that runs on a binary search tree as **BST-Select**, which is given a rank k and must return a pointer to the node in BST with that rank.

Assume that we are given a binary search tree, and we initialize a pointer x to be the root of the tree. The algorithm works by searching down a path of the tree using the variable x, until we "land" on the element with rank k. How do we decide if we should search to the left or the right for our element of rank k? Or indeed if we have already found the correct element? We examine the following three scenarios:

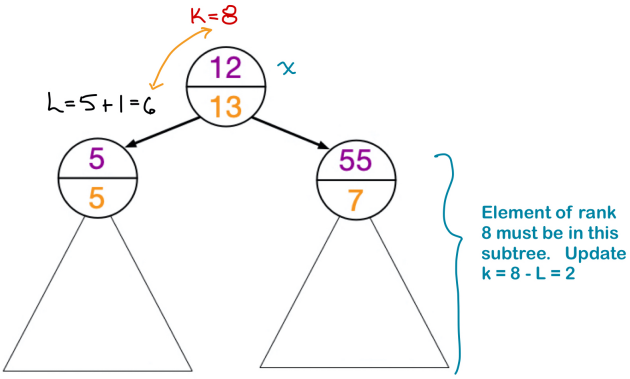
- **Case 1:** Let L be the number of items in the *left* subtree, *plus* one. If it happens that $L = k$, then in fact we have found the node x of rank k. An example is shown below:



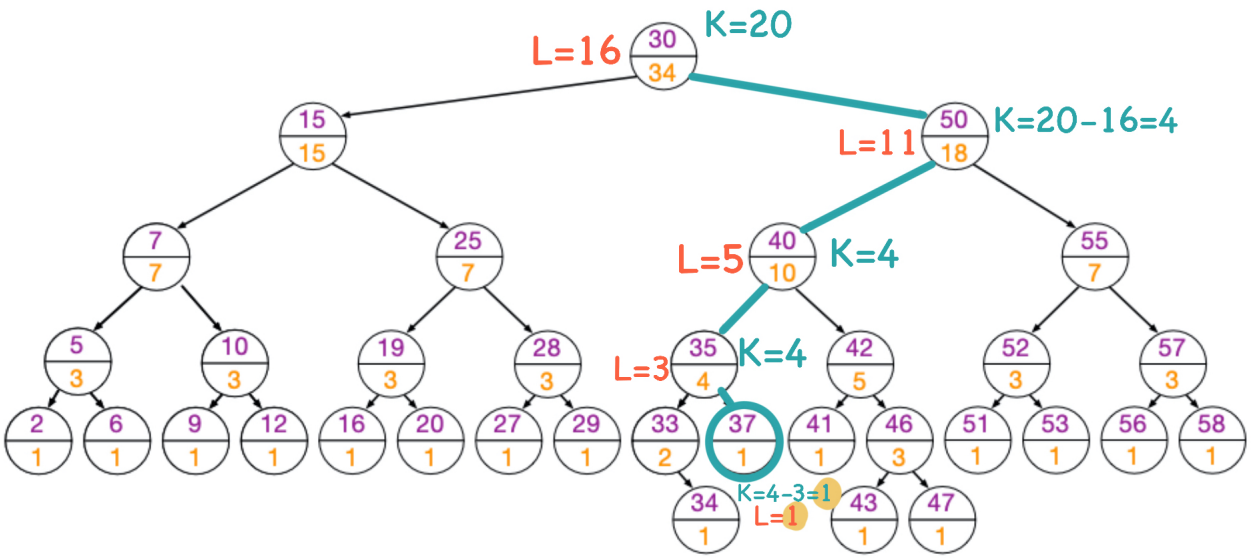
- **Case 2:** If $k < L$, then the item we are searching for must actually be in the left subtree of x. In this case our search should continue to the left. The variable x will now move to the left child ($x = x.left$) and we repeat our process on the left subtree. An example is shown below:



- **Case 3:** Finally, if $k > L$, then we continue the search to the right by letting $x = x.right$. We must also update $k = k - L$ to compensate for the L items that are smaller than those in the right subtree. An example is shown below:



A complete example of a search for rank $k = 20$ is shown below:



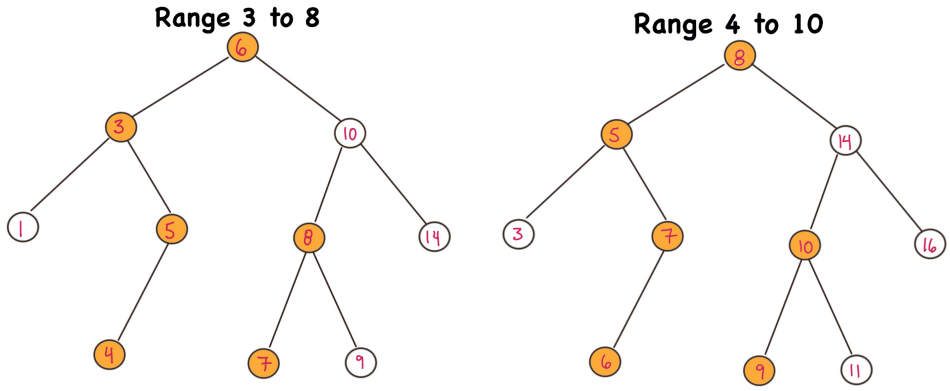
The algorithm can be implement both recursively and iteratively. The pseudocode for the Rank algorithm below provides an iterative solution, performing a while loop that traverses the tree until it finds the item with the appropriate rank (Case 1) in which case the algorithm terminates and returns x. The initial parameter x points to the root of the tree. The algorithm below assumes that an element of rank k exists in the tree.

```
BST-Select(k,x)
    L = 1
    if (x.left ≠ NIL)
        L += x.left.size
    while (L ≠ k)
        if (k < L )
            x = x.left
        else
            x = x.right
            k = k - L
        L=1
        if (x.left ≠ NIL)
            L += x.left.size
    Return x
```

Runtime: BST-Select will eventually halt when it reaches the item of rank k. The runtime of each step through the above loop is constant, thus the runtime is $O(h)$. For a Red-black tree or AVL tree, this has runtime $O(\log n)$.

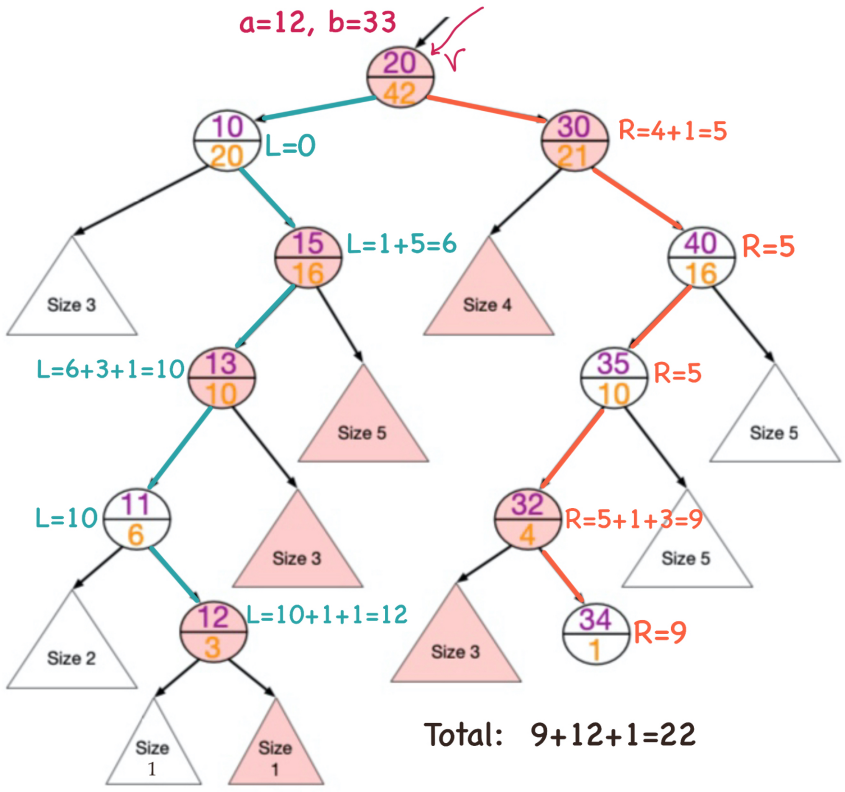
4 Ranges

Our next operation is called Range(a,b), and it returns the **total number of elements** that are in the **range** $a \leq x \leq b$. Note that the endpoints of the range do not have to be in the tree. For example, the tree on the left below has exactly 6 elements in the range $3 \leq x \leq 8$, and the tree on the right also has 6 elements in the range $4 \leq x \leq 10$, even though the element 4 is not in the tree.



The algorithm for finding the number of elements in this range is very similar to the *Rank* algorithm from above, although now the process is repeated twice. We describe the tree main steps of the *Range* algorithm:

- **Step 1:** Start at the root of the BST and perform a search down the tree until we find the **first** element that is in the range $a \leq x \leq b$. Call this node v:
- **Step 2:** Starting at node v we perform a tree search for key a. Similar to the *Rank* algorithm, we keep a running total in the variable *left-total* that represents the number of items in the left subtree that are greater or equal to a. This *left-total* is found by summing the **right subtree sizes** on the path to a, and counting the elements on the path the a that are greater than a. The search terminates when we find a key with value a, OR we arrive at a leaf.



- **Step 3:**. Starting at node v, we perform a tree search for key b. We keep a running total in the variable *right-total* that represents the number of items in the right subtree that are less than or equal to b. This *right-total* is found by summing the **left subtree sizes** on the path to b, and counting the number of elements on the path to b that are smaller than b. The search terminates when we find a key with value b, OR we arrive at a leaf.

  When all steps are complete, the final value to be returned by the Range algorithm is:

$$\text{left-total} + \text{right-total} + 1$$

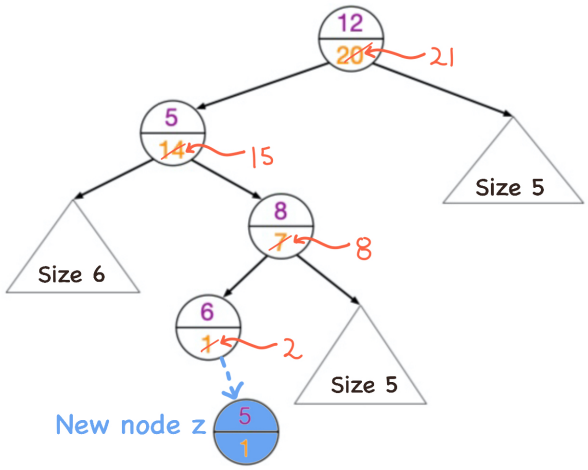
Runtime: Each search in a balanced binary search tree runs in time $O(h)$. The calculation of the totals only ads a constant step per node on the search path. Therefore the total runtime remains $O(h)$. As usual, if the tree is a Red-Black tree or AVL tree, the runtime is $O(\log n)$.

5 Inserts and Deletes:

Augmenting the binary search tree with subtree sizes enables us carry out new operations in time $O(h)$. However, we must verify if maintaining this information in the tree is feasible. What happens if we carry out an insert/delete operation? The subtree sizes are changed, and it is important that we can quickly update this information in our new augmented tree. The goal is to make sure that any updates that we must do to the augmented information do not substantially change the runtime of the Insert and Delete in a BST.

5.1 Inserts:

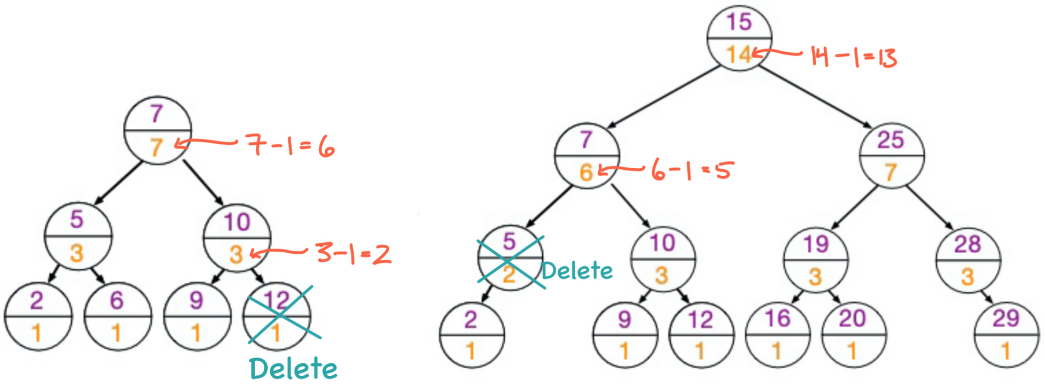
Let's first think about what happens when we insert a new node into the tree. All of the subtrees on the path from the root to x will get *larger* by one. Therefore as we search down the path to x, we can simply increase each subtree size on the path by 1:



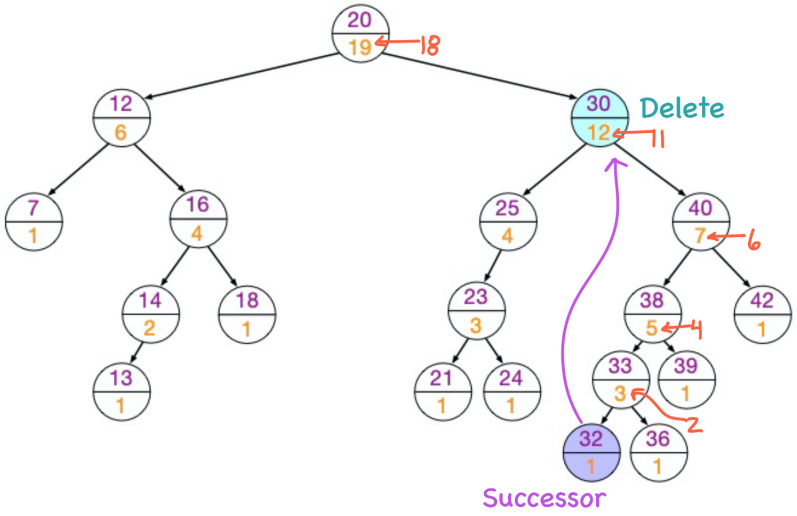
Runtime: This update takes only a constant amount of time per node on the path to x. Therefore the insert operation remains $O(h)$ in the augmented BST.

5.2 Deletes:

Recall from our section on BSTs that there are three cases for the deletion of node z in a binary search tree. The first case is when the deleted node has *no* children, and the second case is when the deleted node has *one* child. In both of these cases, after the deletion, we must **decrease** the subtree sizes for all nodes on the path from the root to z. The diagram below shows an example of the subtree sizes that must be updated after a node is deleted with either no children (left) or one child (right).



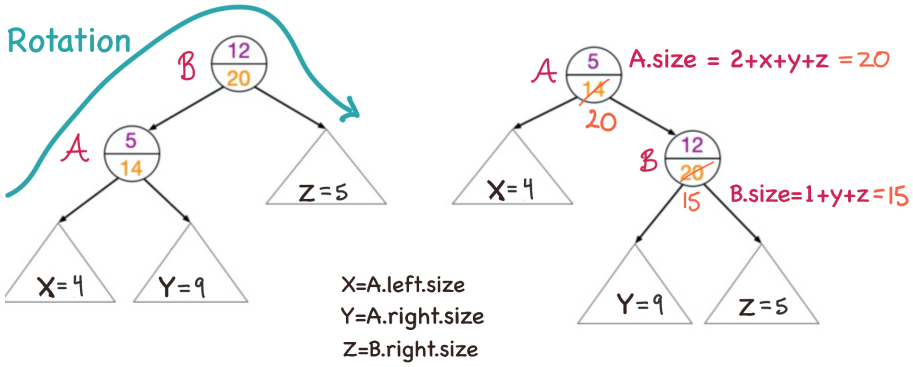
If the node had **two** children, then the deletion is made using the successor node. In this case, we decrease the subtree sizes for all nodes on the path from the *successor* to the root.



Runtime: This update takes only a constant amount of time per node on the path to z. Therefore the delete operation remains $O(h)$ in the augmented BST.

5.3 Rotations:

Unfortunately, we may also make structural changes during an insert/delete operation. Both red-black trees and AVL trees rely on rotations to keep the tree balanced, and rotations *change* the subtree sizes. Therefore we must also be able quickly update the subtree sizes anytime we make a rotation. We show here that updating the subtree sizes during a rotation is also a *constant-time* operation. Suppose we carry out a right rotation as shown below. Assume that the three subtrees have sizes x, y and z. Notice that after the right rotation, the subtree sizes of the triangles below *do not change*. However the subtree sizes of A and B **do** change. We can simply update $A.size$ to $2 + x + y + z$ and $B.size$ to $1 + y + z$.



Runtime: Updating subtree sizes during a rotation is constant, and therefore the overall runtime of the insert/delete operations in a balanced BST remains $O(\log n)$.

6 Summary

We have shown that by storing the subtree size in the node of a binary search tree, that we can carry out BST-Select(k) and $Rank(x)$ in $O(h)$ time. Furthermore, updating the subtree size information during an insert and delete can also be done in $O(h)$ time. Finally, if the tree requires rotations (such as Red-black or AVL) during an insert or delete, we can update the subtree sizes accordingly in constant-time per rotation. Therefore inserts and deletes in red-black or AVL trees still run in time $O(\log n)$.